

Tamper Detection Marking for Object Files

Mike Jochen and Lori L. Pollock
University of Delaware
Newark, DE 19716, USA
Email: {jochen,pollock}@cis.udel.edu

Lisa M. Marvel
U.S. Army Research Laboratory
Aberdeen Proving Ground, MD 21001, USA
Email: marvel@arl.army.mil

ABSTRACT

Much of present day computer software is highly mobile, with a great amount of software being delivered to a client host via a network shortly before execution begins. The integrity of mobile code is one important aspect for the secure execution of the code on the client host. We describe a cryptographic-steganographic approach to embedding authentication data within executable object files. Our approach simplifies management of and requires no additional bandwidth to accommodate the authentication data. Initial experimental results show no runtime performance degradation for the execution of the protected program.

INTRODUCTION

Programs today are highly mobile; updated instructions can be delivered to remote sensors on a wireless network, executable attachments appear frequently with email, and software is downloaded from inter/intra-nets shortly before execution begins, among other examples. These activities have stirred a great deal of research in the area of mobile code security [1]. The fact remains that a newly arrived program to a local host has the potential to inflict a great deal of damage to the local host. The new program could have arrived from a charlatan host or have been modified by a malicious host during transit. In light of this knowledge, security models that address mobile code remain in high demand.

We have developed a system which enables mobile code users to validate executable object files with authentication data that is derived from the code itself, without the risk of separation of authentication data from the

code, without increasing bandwidth requirements, and without relying on a third party authentication agent. Our system embeds authentication data which we term a Tamper Detection Mark (TDM), a cryptographic checksum, within the code as a way to address the issues of code integrity and authentication. This method can be utilized to detect virtually any degree of tampering or alteration to an object file. Authentication with our system is optional; a code carrying a TDM is semantically equivalent to the original version of the code. The TDM-protected code can execute without any special pre-processing should authentication of the code not be desired or should the code execute on a system without TDM validation capability.

Encrypted authentication data in our system is communicated via steganographic techniques [2] which (1) prevent separation of authentication data from the code thus eliminating the situation where retransmission of data is required and (2) decrease bandwidth requirements. Steganography is the process of hiding secret information in other innocuous cover information. The hiding process usually exploits randomness or noise within the data of the cover medium to hide the secret.

The remainder of this paper is organized as follows. Section 1 provides context for this work by reviewing existing security techniques for mobile agents and mobile code. In Section 2, we present an overview of our system and details on its implementation. Section 3 evaluates our system through empirical study. In Section 4, we summarize our results.

BACKGROUND

There are a number of techniques that address tamper detection/protection for mobile code. Tan and Moreau modified the concept of execution tracing to address mobile code security [3]. Under this scheme each host traces its execution of a mobile agent. When the agent wants to move to a new host, the trace can be verified for correct execution via a verification server. Trust in

This material is based upon work supported by the National Science Foundation under Grant No. CCR-0219559.

Prepared through collaborative participation in the Communications and Networks Consortium sponsored by the U. S. Army Research Laboratory under the Collaborative Technology Alliance Program, Cooperative Agreement DAAD19-01-2-0011. The U. S. Government is authorized to reproduce and distribute reprints for Government purposes notwithstanding any copyright notation thereon.

specific hosts can be revoked under their distributed verification system.

Spinellis [4] introduced the idea of utilizing reflection (the ability of a computation process to reason about itself) to provide a mobile code system with the ability to validate the integrity of deployed code on portable devices. With reflection, the code has the ability to “see” its own instructions and respond to queries about its state to some authentication server.

Hohl described a method to protect mobile agents from malicious hosts [5] for a limited time within a “black box”. This technique is commonly known as program obfuscation. Obfuscation attempts to obscure program meaning, actions, and data from a potentially malicious host by introducing overly complex control structures within the agent’s program instructions and by re-composing the agent’s data.

Checksums and digital signatures like MD5 and SHA hash values, or RSA signatures are popular forms of integrity verification for data transmission. Clients can download the desired software and a separate checksum for comparison on the local host. Discrepancies between the downloaded value and the local checksum can indicate potential tampering or transmissions errors.

Our work presented in this paper builds on concepts embodied in our MOST system, designed to embed authentication data within Java class files [6]. MOST utilizes steganographic techniques to embed authentication data within the constant pool table of a class file. While some of the same basic concepts for MOST are applied to this work, the structure and content of a class file differ significantly from that of a native object file, introducing new challenges and approaches.

THE TAMPER DETECTION SYSTEM

In this section, we present an overview of our system and the details involved in creating, embedding, extracting, and validating a TDM.

A. Overview of TDM Framework

An overview of the framework for our system is shown in Figure 1. The system operates in three phases, a *Pre-process Phase*, an *Embed Phase* and a *Validate Phase*. The *Embed Phase* typically takes place on the host which compiles the source code and produces the object file. We shall call this host the trusted host. The *Validate Phase* takes place on the local host desiring to validate the integrity of the code (typically after download and before execution of the application). The same *Pre-process Phase* is required before both the *Embed Phase*

and the *Validate Phase*. A basic example demonstrating our system follows: the trusted host compiles a program, embeds a TDM within the object file of the program, and makes the program available for download. The local host downloads the program, validates the TDM within the object file via our system and proceeds with execution based on the validation result.

The basic concept behind the system is to manipulate the order of relocatable blocks of instructions to encode a value. A relocatable block is similar to the basic block commonly used within the area of program analysis. A basic block is a linear sequence of contiguous instructions which have a single point of entry and a single point of exit for program control flow. The concept of a relocatable block builds upon a basic block in the following way: a relocatable block is a contiguous sequence of instructions which are terminated by a `goto` instruction (i.e., any type of unconditional jump instruction which does not store a return address). Relocatable blocks can have multiple points of entry, and exit, but are always terminated with an unconditional jump which does not store a return address. Thus, a relocatable block can be composed of multiple basic blocks.

1) *Embedding the TDM*: The requisite pre-processing of the object file from Figure 1 is detailed in Algorithm 1. First, machine instructions of the program are analyzed to construct relocatable blocks of code by program disassembly. Once the relocatable blocks have been constructed, the blocks are sorted according to number of instructions and instruction type to begin transforming the program to canonical form. Validation of a TDM is based on program form. Embedding a TDM within an object file changes program form, thus we must be sure that both the trusted host and the local host begin with a program of identical form. The canonical form of the object file meets this requirement.

Moving a block of machine instructions to a new address can have a large impact on the remainder of an object file. Many sections within the file (to include the text section itself) must be updated to reflect the new address of the relocated block of code. All control structures, look-up tables, and address calculations must be appropriately updated with the new address information or the resulting object file will either fail to execute or worse yet, execute with incorrect results. The program in canonical form must be a valid program, a program which can execute correctly on a regular machine with no special pre-processing.

As shown in Algorithm 2, after the object file has been pre-processed, the TDM is created by computing a hash

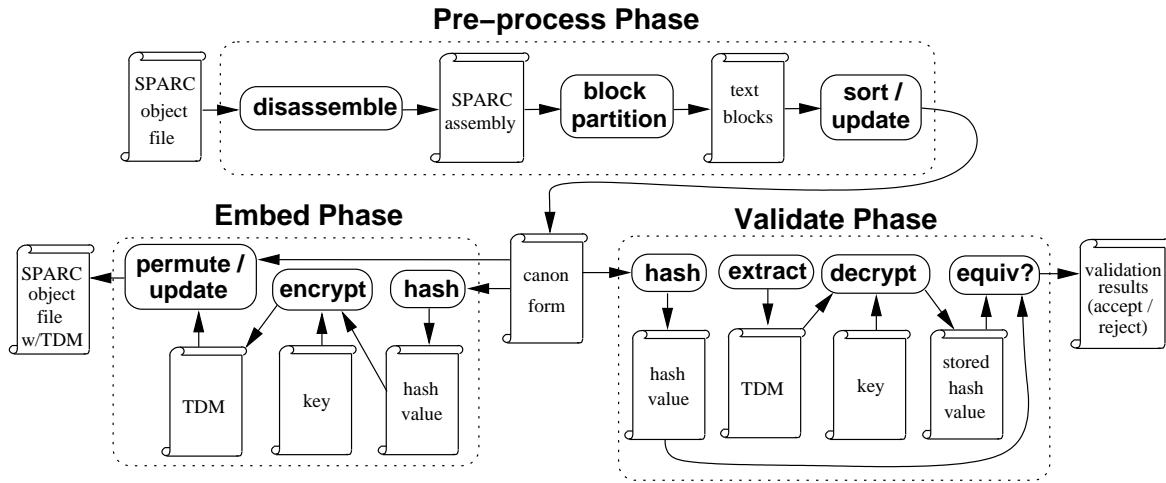


Fig. 1. TDM Framework.

value of the object file in canonical form and encrypting that hash value with the secret key. The TDM serves as a cryptographic checksum for the program.

Algorithm 1 Pre-process an object file.

Input: Object file, O;
 Program counter, PC, points to first instruction of O;
 Program exit, PE, points to last instruction of O;
Output: Array, B, of relocatable blocks of instructions;
 Object file, O, in canonical form;

```

1: /* Build array of relocatable blocks */
2: b_ctr ← 0; /* Initialize block counter to zero */
3: Array B; /* Array to hold blocks */
4: while PC ≠ PE do
5:   b_start ← PC; /* Begin new block */
6:   while decode_instruction(PC) ≠ 'goto_instr' do
7:     PC ← PC + 1;
8:   end while
9:   B[b_ctr] ← newblock(b_start, PC); /* Append block to B */
10:  b_ctr ← b_ctr + 1;
11:  PC ← PC + 1;
12: end while
13: /* Transform object file to canonical form */
14: sort(B); /* sort relocatable blocks */
15: foreach section S, in O do
16:   if S contains instruction addresses then
17:     update addresses in S to new target addresses;
18:   end if
19: endfor
20: /* return array of relocatable blocks and canonical form */
21: RETURN (B, O);

```

The actual process of embedding the TDM in an object file is accomplished by permuting the order of the previously found relocatable blocks of program text. The permutation algorithm utilized is that given by Knuth [7]. To reorder the sequence of blocks of machine instructions to encode a TDM, we select the n^{th} permutation

Algorithm 2 Embed TDM into an object file at the trusted host.

Input: Object file, O, in canonical form;
 Array, B, of relocatable blocks;
 Encryption key, K;
Output: Object file, O, with embedded TDM;

```

1: hash_value H ← hash(O);
2: TDM ← encrypt(H, K);
3: permute(B, TDM);
4: foreach section S, in O do
5:   if S contains instruction addresses then
6:     update addresses in S to new target addresses;
7:   end if
8: endfor
9: RETURN O; /* object file with TDM */

```

of all possible orderings of those blocks of instructions. Permuting the order of program text blocks requires the entire object file to be updated again to reflect the final sequence of instructions. The embedded TDM is now part of the object file, represented by the ordering of machine instructions.

Algorithm 3 Validate TDM from an object file at the local host.

Input: Object file, O, in canonical form;
 Array, B, of relocatable blocks;
 Encryption key, K;
Output: Boolean (T/F) if TDM is validated;

```

1: hash_value H ← hash(O);
2: TDM ← permuteInverse(O, B); /* Extract TDM */
3: decrypted_hash DH ← decrypt(TDM, K);
4: RETURN (H = DH); /* Equal if TDM is valid */

```

The program is ready for delivery to the local host once the TDM has been embedded within the object file

by Algorithm 2. The new object file created during the *Embed Phase* is semantically equivalent to the original object file; the same computation is performed and the runtime performance of the computation should not be adversely affected. This new object file with embedded TDM must be able to execute on a regular machine with no requirement for special pre-processing.

2) *Validating the TDM*: Once the program is received from the trusted host, the local host can validate the TDM with our system. The TDM of the object file is extracted by executing the *Pre-process Phase* followed by the *Validate Phase* given in Algorithm 3. The first steps of validating a TDM are similar to embedding a TDM; the object file is pre-processed to achieve canonical form, and a local hash value of the object file is computed. The TDM is extracted from the object file by reversing the permutation algorithm used during the *Embed Phase*. The sequence of relocatable blocks in canonical form is compared to their permuted sequence to determine which of the $n!$ permutations is encoded in their ordering. The extracted TDM is decrypted with the secret key and compared with the locally computed hash value. If the program has not been altered since insertion of the TDM and the proper keys have been used to create and validate the TDM, the validation result will return true. A failing *Validate Phase* will result from any alteration to the program or incorrect key use during generation/validation of the TDM.

B. Security Properties

The security of our system is reliant upon the security of the encryption and hash functions used to create the TDM (and the security of the key distribution and management by those using the system). As long as the hash function employed by our system has a low probability of collision, the probability of an opponent finding another program that has the same hash value as the original code is low. Compounding the difficulty of a collision attack in this unlikely event is the fact that the program that collides with the program under attack must contain legal machine instructions, or the program will fail to execute. A desirable property of any good hash function provides for a change in 50% of the bits of the hash value should there be a 1-bit change to the hashed file. For our prototype implementation, we used two well-known hash functions, MD5 and where possible, SHA-1. The encryption algorithm used is 3DES as it is a readily available shared key algorithm from the Botan cryptographic library [8]. However, our approach has been designed to permit both the hash

TABLE I

TIME TO EMBED AND VALIDATE A TDM IN UNOPTIMIZED CODE.

Benchmark	Size	Lines	Blocks	Embed	Validate
wave	7.0	26	17	0.102	0.098
sort	7.5	211	17	0.099	0.099
bmm	8.8	76	21	0.102	0.101
wc	9.2	167	26	0.102	0.099
paraffins	11.0	287	57	0.108	0.109
compress	83.0	1431	97	0.151	0.151

and encryption functions to easily accept new algorithm implementations as desired.

Our system will always be able to extract a mark from an object file that contains an executable text section; there is always a sequence to the order of instructions. Thus, separation of authentication data from the code is not possible; removing the mark requires tampering with the code, which is detected by our tool. The extracted mark from this code will either be a valid TDM or garbage (e.g., the TDM may have been scrambled by disassembly/recompilation yielding an invalid mark). If the extracted mark is a valid TDM, the system will correctly validate the program. If the mark is garbled, the system will identify the program as altered.

Our system is very sensitive to changes to TDM protected code. Even if an altered program is semantically equivalent to the original program (e.g., slight changes have been made to the object file which do not affect the computation), the system will signal a validation error. Any alteration found by hash disagreement, no matter how insignificant, is detectable by the system.

Our system assumes the secret key for the TDM is transmitted over a secure channel prior to transmission of the mobile code. While key distribution and management is critical to security interests, this work does not attempt to address this problem. Key distribution and management is an area of active research with several worthwhile treatments and analyses. A good starting point for the reader can be found in [9], [10].

EMPIRICAL EVALUATION

A prototype system was coded in C++, utilizing the Botan cryptographic library [8] for all hash/cryptographic algorithms. All experiments were conducted on a 300MHz Sun Ultra-10 machine running Solaris8 with 192 MB of physical memory. All reported times are the median of five execution runs.

Table I shows times for embedding and validating a TDM for several benchmarks. The benchmarks presented in Table I have been compiled without optimization. The

TABLE II

TIME TO EMBED AND VALIDATE A TDM IN OPTIMIZED CODE.

Benchmark	Size	Lines	Blocks	Embed	Validate
wave	6.5	26	4	0.102	0.097
sort	6.7	211	8	0.098	0.096
wc	8.4	167	19	0.101	0.100
bmm	8.6	76	10	0.101	0.099
paraffins	9.3	287	18	0.103	0.100
compress	80.0	1431	50	0.114	0.113

size of the benchmark is given in KB as well as number of lines of source code (less comments). The number of relocatable blocks identified in the text section of the benchmark is also given. The time required to embed (including pre-process and create) a TDM as well as to validate (including pre-process and extract) a TDM is given in seconds. Table II shows the corresponding times for the same benchmarks which were compiled with common program optimizations enabled. The current prototype implementation of this system targets SPARC binaries compiled via gcc. The focus on SPARC binaries from a single compiler eased implementation issues in the design of our proof of concept system.

Note that it did not take much time to process each benchmark (ranging from 0.096 seconds to 0.151 seconds for either embed or validate phases). Furthermore, no runtime penalty was observed for TDM protected codes (i.e., execution time remains constant from original to TDM-protected programs). This was a point of interest to our work as relocation of machine instructions could impact runtime performance. We hypothesize that since we are relocating only those blocks of code which are always terminated by an unconditional jump, there is minimal interference with compiler optimizations and minimal disturbance to original runtime performance.

As MD5 is a 128 bit hash algorithm and DES is a 64 bit cipher, two 64 bit blocks are required to store the encrypted hash value (TDM) in the object file. When SHA-1 is employed, the 160 bit hash value is padded to 192 bits for three 64 bit blocks. Should the object file not contain enough relocatable blocks to encode the minimum 128 bits (35 blocks) the remaining bits are hidden in other areas of the file which will not increase file size (e.g., empty pad areas, unused bits of special instructions, etc). This is a special case required for rather small programs only. We are actively searching for other areas of the object file which exhibit the required traits to accommodate steganographic manipulation.

CONCLUSIONS

We have presented the design and implementation of a system capable of communicating program authentication data within executable binary code in a manner which simplifies management of the authentication data and is bandwidth efficient. With our system, a Tamper Detection Mark is encoded within an object file via steganographic techniques. This process eliminates the risk of data separation and additional bandwidth requirements of conventional validation techniques, making our system desirable in areas where bandwidth is limited (e.g., low power, wireless networks). A user of this system choosing to validate a TDM-protected program is assured that the program was certified by a host holding the proper key and that the program was not altered in any way between the time the program was marked with the TDM and the time the user performs the validation check. Analysis indicates that our system detects with a high degree of probability, any tampering with an object file and can do so within a reasonable amount of time.

“The views and conclusions contained in this document are those of the authors and should not be interpreted as representing the official policies, either expressed or implied, of the Army Research Laboratory or the U. S. Government.”

REFERENCES

- [1] G. McGraw and G. Morrisett, “Attacking malicious code: A report to the Infosec Research Council,” *IEEE Software*, vol. 17, no. 5, pp. 33–41, Sept./Oct. 2000.
- [2] S. Katzenbeisser and F. A. P. Petitcolas, *Information hiding techniques for steganography and digital watermarking*. Artech House, 2000.
- [3] H. K. Tan and L. Moreau, “Certificates for mobile code security,” in *Proceedings of the ACM symposium on applied computing*. 2002, pp. 76–81.
- [4] D. Spinellis, “Reflection as a mechanism for software integrity verification,” *ACM Transactions on Information and System Security (TISSEC)*, vol. 3, no. 1, pp. 51–62, 2000.
- [5] F. Hohl, “Time limited blackbox security: Protecting mobile agents from malicious hosts,” in *Mobile Agents and Security*, ser. Lecture Notes in Computer Science, G. Vigna, Ed. Springer-Verlag, 1998, vol. 1419, pp. 92–113.
- [6] M. Jochen, L. Marvel, and L. L. Pollock, “MOST: A tamper detection tool for mobile java software,” in *Proceedings of the 3rd Annual Information Assurance Workshop*. IEEE, 2002.
- [7] D. E. Knuth, *The Art of Computer Programming: Seminumerical Algorithms*. Addison-Wesley, 1981, vol. 2, 2nd ed.
- [8] “Botan crypto library,” <http://botan.randombit.net/>, 2002.
- [9] V. Shoup, “On formal models for secure key exchange,” IBM Research, Report RZ 3120 (#93166), 1999.
- [10] R. Canetti and H. Krawczyk, “Analysis of key-exchange protocols and their use for building secure channels,” in *Advances in Cryptology – EUROCRYPT*, B. Pfitzmann, Ed. Springer-Verlag, 2001, pp. 451–472.